Automatic Coding for Business Applications

By R. M. Paine

Automatic coding is defined, and the different elements described. The ideals for business applications are considered and some of the difficulties discussed. The concept of a common source language is put forward and COBOL is used as an example of the structure of common language for business purposes. This paper was presented at the Harrogate Conference of The British Computer Society on 5 July 1960 and refers to the position at that time.

Introduction

Great interest is being shown in the U.S.A., the United Kingdom and Europe about automatic coding, and this paper will offer a definition of automatic coding for business applications, and then state some of the reasons why computer people are paying so much attention to this subject.

Definition of Business Automatic Coding

As we well know, systems planning and programming for commercial work involves much detailed, laborious work, and requires several disciplines and languages—those of the accountant, the analyst, the programmer, the computer, etc. Automatic coding for business problems is a system by which a computer can be given a series of statements in simple English describing a complete business operation. The computer, by means of a master program or compiler, can be used to translate these statements into the machine's own code, and will allocate storage and produce a program for the job. This program, when run against the data for the job, will produce the desired results such as printing invoices, accumulating statistics, checking totals, updating files, and so on.

Source Language

The statements in English which form the procedure to be followed by the computer are called the *Source* language, the program in machine code finally produced is called the *Object* language, and the master program which carries out this work is called the *Compiler* or *Processor*.

The Source language describing the procedure consists of:

(*a*) Verbs or action words which call into operation routines in the Processor, and specify the translation required. These verbs have the same meaning in all applications or user companies.

(*b*) Nouns or names for files, records, fields, etc., which would vary with the company using the system.

(*c*) The logic or syntactical rules for forming statements and expressing decisions. These rules would have the same meaning in all applications or user companies.

Data Description

Two other elements, closely connected with the source language and which can be considered part of it, are the data description and the environmental considerations.

The nouns or names which have been used in the statements have to be defined so that the compiler knows where and how to obtain the information involved in input, processing, and output, and in exactly what layout, radix, size, etc., the data are stored. This information is normally presented to the compiler separately from the procedure statements. The data description section is a very important part of automatic coding and requires ingenuity in its design to prevent it becoming too irksome in length or some important fact being overlooked.

Environmental Factors

The processor may also need to be informed what equipment or computer specification is to be used on the actual runs of the business application. This information includes points such as that certain files are on magnetic tape, that others on cards, how many printers are available, what size internal store is to be used, etc. This environmental description will be especially important in the concept of a common source language.

The processor using the procedure statements, the data description, and the environmental factors, will translate the English into the computer's own code, in one or more runs of the statements, and produce the object program. The translation is of course a once only job, since once a proved object program has been produced this is used for the daily, weekly, or monthly running of the job.

Ideals of Business Automatic Coding

What advantages do people think they will gain from business auto-coding? There appear to be about six main considerations.

Firstly, from the point of view of the accountant it gives hope of a computer language that he can understand and therefore use to control the procedures that are prepared for a computer system. The systems analyst would also like to write the procedure statements to tell the computer how to tackle the job, but at the moment he may shudder at the brink of learning machine coding,

or even the notation of programmer's flow charts. Thus, with an automatic coding system, there is hope of the computer being instructed by the people responsible for the job at present, and of management being able to understand what the "beastly machine" is trying to do.

Reductions in Cost

Secondly, there are expectations that the cost of programming and the time taken to put an application on to a computer will be considerably reduced. This expectation arises from several factors: (*a*) there will not be so many stages or disciplines to go through to obtain the successful running of a commercial computer system: (*b*) less hours will be required for writing the program or procedure statements since fewer steps or detail will be required, and the time-consuming job of allocating storage, avoiding the overwriting of program, etc., will be reduced if not eliminated: (*c*) perhaps less staff, or even lower-calibre staff, may be used.

There are few figures available as to the extent of the cost reduction or even of the present programming and systems cost. But some American experience is said to point to the fact that getting a commercial system on to a computer costs as much as the hardware itself. Thus any possible reduction in this immense cost is very welcome.

The third advantage, which is very closely connected with the second, is that de-bugging time and trouble should be drastically cut down. This arises of course only if the source language statements are logically and procedurally correct in the first place. But automatic coding should be able to help in this extremely irritating area, for both programmers and operators, of dealing with minor coding mistakes and foolish errors in the program. It seems that compilers, provided they have good print-outs so that any errors that do occur can be easily traced and remedied, should be able to prepare object programs without the re-occurring normal human frailties.

Easier Adaptation

A fourth ideal is that programs will be easier to alter with changing circumstances, since a program is seldom finished and unalterable for a business application. Management is always calling for further information: or a new deduction, as in the graduated pension scheme, arises in the payroll: or a new product is added to the line. The object program need not be modified, which would call for a detailed knowledge of machine code, but instead the English language statements would be changed, and that part, or the whole, of the source language re-compiled to produce a new object program.

Fifthly, it is hoped that training of staff will be quicker, since less people will need to know the details of the machine and its code, and can concentrate instead on the problems to be tackled and the easier English statements. If training can be given in a few days rather than weeks, one-off jobs could be as feasible in commerce as they are in science.

The sixth ideal is concerned with a common language which can be used to prepare programs for all types of computers, provided that each computer has had a special compiler written to accept the common source language. This could almost be conceived as the final aim of evaluating committees—to test all machines on their company's own work before making a decision! A common source language would also be useful for a company in reducing the cost of re-programming in the event of replacing their computer installation by another, because of the expansion of the firm or an extension of the activities to be processed by a computer system.

Difficulties and Drawbacks

The foregoing seems to present a picture of the never-had-it-so-good accountant and systems man. But what are the problems which may mar this image? Again, there are about six main points which require consideration.

To start with, no matter what people claim or think, there is still a vast amount of detailed, lengthy work to do in rethinking and putting a business problem on to a computer. Automatic coding may help the actual coding side, but months or years of systems work will still be required to find out what we want the source language statements to say. Mr. J. J. Finelli, of the Metropolitan Life Insurance Co. of New York (who have four large Univac Computers in their head office), said in his recent lecture (Finelli, 1960) to The British Computer Society: "The matter of developing codes for a computer program, however, is not the biggest part of the job. Any assembler can come into play only after you have made up your mind as to what the program should do. Developing the specifications is the more difficult part of the job."

We should not, therefore, imagine the chief accountant or his assistant sitting down and straightaway writing a statement of the application in English which will be presented to the machine for compiling. Life is not as simple as that! When people have spoken of the systems analysis and programming costing as much as the hardware, by far the greater part of the cost has been on the systems study, not the programming, and this will still remain.

Formalized and Strict English

Secondly, in the writing of the English statements of the source language, discipline and skill will still be required. True, the processor will take care of loops, modifications, end-of-tape procedures, etc., and produce an object program using a consistent strategy, so that all programs produced for an organization will abide by the same techniques, of coding—a house style in effect. But the systems analyst or programmer writing the statements will still have to understand the rigorous logic and format of his language. This may be no easy matter (as later examples perhaps show) and in no sense

is it a matter of a person describing a procedure in his own normal English. He must know the permitted verbs, the nouns, the way of stating conditions, how to refer to another section, and so on.

You have probably had difficulty in reading descriptions of automatic coding source languages, and their restrictions and problems of syntax. A lively American magazine, *Computing News* (Computing News, 1960), recently had an amusing satire on the work of an imaginary committee working on a source language called POOGOL:

"The world watched with bated breath while the shapers of POOGOL debated the limits to the number of times a subroutine was to be permitted to call on itself, and many hours of heated debate were spent on resolving the question of whether certain quantities in a restricted class of paralyzed variables were to have 'names' or 'titles.'"

These points are important, however, and serve to emphasize how more difficult it may be for a non-programmer to pick up the source language than he expects.

A third difficulty, leading on from the previous one, is that the statement writer may still have to know something about the computer to be used and perhaps, for the most efficient object program, about the compiler. The compilers, at present, do not plan the number of runs required for a job, or what should be done on each run. The systems analyst must know sufficient about the capacity and functioning of his computer to decide whether the cost analysis, say, can be performed in the internal store, or whether the data needs to be output, sorted and re-input. He must decide, for instance, the sequence of all the files, that the first run will produce the payroll and payslips, the second run will produce an order and expense analysis from data produced in the first run, and that a preliminary run will be required to amend files in respect of permanent changes. (It is this difference of machine capacity and specification which may prove difficult for the idea of a common language.) The analyst then has to write the source statements of computer activity for each run. He does not, under present systems, just describe his problem in terms of inputs and outputs and present it to the computer. Instead he must initiate a method of solving the problem and present a series of steps to the computer, telling the machine what to do, in a sequence best suited to the computer.

Not Accountants' Language Yet

Fourthly, there may be a disappointment in store for accountants who have hoped that automatic coding would use their accountancy language rather than codes such as, for example, 60 for clear-add, and 55 for left shift. Source languages certainly get away from the 60's, the 55's and so on. But the meaning of accountancy terms such as "post," "invoice," "balance," etc., varies so much from firm to firm that the verbs used in source languages have to be rigorously defined and smaller in scope. The exact accountancy practice required has to be built up from these brick verbs, otherwise a source language might be of use to only one or two firms. The verbs used in the source languages tend to be "Compare," "Read," "Edit," "Close," "Add," from which the precise meanings of "Post" can be constructed. Accountants and management in general may be able to read the prepared English statements and gain an overall impression of what is being done, but it is hardly their function to construct the statements. Such comments as "Management can also learn quickly how to prepare their own programs, gain complete control over the data processing, and no longer become dependent on the programmers" (Mitchell, 1960), are exaggerated and give a wrong impression to management.

"Noise" Words

A fifth difficulty is that Data Descriptions and Statements must be exactly right: no wrong commas, no colons instead of semi-colons, no incorrect indentations for various levels of statement or record, no mis-spellings, no unassigned over-punching, etc. This can be very tiresome and cause a lot of writing and re-writing of the procedure statements. For instance, look at the following conditional statement and think how easy it would be to get the punctuation or phrasing wrong:

"If *X* equals *Y* then move *A* to *B*: otherwise add *A* to *C* and also if not positive, go to Error Routine: If *J* is greater than *K* perform routine 1 through routine 5 and add *J* to *K*: otherwise add *A* to *B*":

Not only must the punctuation, etc., be correct, but the procedure statements will also be prone to error because they are so "wordy" due to the presence of "noise" words. "Noise" words are those words used to improve the readability of the language but whose presence or absence does not affect the meaning of the statement or the action of the compiler. A cynic might perhaps define a computer conference as made up of contiguous "noise" words. An example is the phrase "If *A* is greater than *B*," where the words "is," and "than" have no significance for the compiler but make the phrase easier to read as English.

This searching for readability tends to make the procedure statements verbose, and as the person writing the statements becomes familiar with them and therefore less concerned with their English flow, he may be driven to some form of shorthand or symbols to escape writer's cramp. The aim would then be to obtain correct punctuation, etc., allow symbols for the programmer's convenience, and yet produce a readable version for the programmer's lord and master to understand. This can probably be done by the use of pre-punched cards with the standard phrases or conditions on them, with blank field names. The programmer could refer to these phrases by symbols, the cards could be pulled, the names or nouns punched, and the cards tabulated to provide a complete English version and a card input to the compiler.

Wasteful of Running Time?

The sixth point concerns the efficiency of the object programs produced by the processors. That is, how long does it take for the object program to perform the application compared with a program written in machine code by a good programmer? The efficiency of an automatic-coded object program is usually described as a certain percentage of the hand program: for instance, an object program 50% efficient means that it would take twice as long to do the actual data-processing job as the hand program.

There are very few facts on this important subject of efficiency since few jobs have been coded both ways; not many automatic-coded jobs have been tackled anyway and, indeed, hardly any compilers have been completed and tested. A further difficulty is the wide range of commercial applications for which automatic coding is supposed to cater, so that an average efficiency would be misleading for a particular application or firm. Mr. Finelli, in the same talk as mentioned previously (Finelli, 1960), thought that Flowmatic which his company had examined would for them take about 20% more running time than the hand-coded programs, and he went on to say: "The supplier, being the supplier of many customers, must think in terms of developing a general system to be used by us and others. As a result many general programs are produced, not programs specially designed for the insurance business. Had we used FLOWMATIC we would have had less difficulty in correcting programs, it does that very nicely. But we would have paid too much for it in terms of running costs."

On the other hand, Remington Rand have stated that, on most problems, near or equal running time should be possible since every part of the program constructed by the compiler has been carried out using the thoughts and care of a first-class programmer concentrating on one particular task at a time, rather than thinking of a job as a whole. I.B.M. have mentioned that with their automatic-coding system on the 709 an efficiency of 95% is achieved in the object program compared with coding in symbolic codes. Current commercial systems such as COBOL expect to obtain at least 80% to 90% efficiency; 90% efficiency would mean that the running time of 40 hours for a job by hand coding might take 44 hours by automatic coding. (Some people would say that over 100% efficiency can be obtained by having lousy hand programmers in the first place.) Until more experience has been gained it is difficult to draw firm conclusions.

The six difficulties and drawbacks mentioned add up to the fact that, after a long struggle to establish the idea of automatic coding for business problems, it may now have been over-sold and over-simplified so that too much is expected of it too soon. Despite the problems, many of them not mentioned here, it has tremendous possibilities. But like the "new, unique, brightening ingredient," which appears in all detergent packets, manufacturers are offering automatic coding schemes with their machines as a matter of course, and almost as a matter of publicity. They should beware lest the users become disillusioned because their clothes are not suddenly, strikingly whiter than before.

Common Language

Having considered the ideals and difficulties of automatic coding for business applications, we can proceed to the idea of a "common source language." The aim of a common language is that a business problem can be described in the source language irrespective of the computer concerned and irrespective of the way in which data is held. The source language program can then be run on any computer, which has the necessary compiler, and an efficient object program produced. As mentioned previously, this could be useful in assessing various machines or in moving from one computer to another.

Since computers do vary in their storage capacities, input output facilities, binary or decimal representation, etc., the Environmental section and possibly the Data section would have to be altered for each computer, but the Procedure statements could remain the same. This is an interesting concept and has received great attention by users and manufacturers. There are difficulties, such as the fact that on different computers the problem might best be tackled in a different way or sequence, and the efficiency of the object program may vary enormously from machine to machine, and from one type of job to another. The feasibility of the idea is best tested by producing a source language and using it on different machines and problems.

Origin of Cobol

While people in the United Kingdom were still wondering whether automatic coding would work for business problems and whether it was too early to standardize a source language, the Americans have gone into action and, under the sponsorship of the Department of Defence (a large user of computers), they have produced a common language, COMMON BUSINESS ORIENTED LANGUAGE or COBOL. It may not be the last word, but at least it is a starting point for future development.

The organizations participating in the original development were:

Air Material Command, U.S.A.F.
ElectroData Division of Burroughs.
David Taylor Model Basin, Dept. of Navy.
I.B.M.
Minneapolis-Honeywell.
National Bureau of Standards.
R.C.A.
Sperry Rand.
Sylvania Electric Products Inc.

In the United Kingdom it has been announced that I.C.T. have adopted COBOL as a source language for the 1301 computer, are adding such things as sterling arithmetic, and are writing a compiler for it. I.C.T. are

already teaching COBOL source language on their training courses for their own and customers' staffs.

Description of Cobol

How can COBOL be described in a few pages when a thick book has been produced and has still left doubts with some people? Obviously one can only give an outline of the source language—the compilers of course will be written by the manufacturers for specific machines. As is required, the Procedure Division is essentially machine-independent (e.g. it says Read File not Read Tape or Read Card) and it is the provisions in that Division that will be mentioned in this paper.

There are 51 characters in the COBOL language, including the digits 0 to 9, the alphabet A to Z, the hyphen or minus sign, the punctuation characters shown on the next line plus space:

" ( ) . , :

and 8 characters to define the operations involved in formulas and relations:

* ** . .

Verbs

COBOL uses verbs to denote actions, sentences to describe procedures, and "If" clauses to provide alternative paths of action. The hierarchy of writing a procedure is EXPRESSIONS, STATEMENTS, SENTENCES, PARAGRAPHS and SECTIONS. There are 23 verbs used, 6 of them for directing the action of the compiler only, and 17 of them for describing the procedure. They are the following, to which "If" can be added as having the function of a verb.

ADD	GO	DEFINE	
SUBTRACT	ALTER	ENTER	
MULTIPLY	PERFORM	EXIT	COMPILER
DIVIDE		INCLUDE	DIRECTING
COMPUTE	MOVE	NOTE	VERBS
	EXAMINE	USE	
READ			
WRITE	STOP		
OPEN			
CLOSE			
ACCEPT			
DISPLAY			

To illustrate the use of the verbs a sentence could be "Subtract issue quantity from stores-record quantity giving new stores-record quantity." The phrase "giving new stores-record quantity" is optional and if left out the result of the subtraction will be left in the "stores-record quantity" field, that is the field mentioned after the "from."

In commercial work some calculation is necessary, and the expression $R = \frac{L(A - C)}{(A + C)^2}$ could be written in COBOL in two forms. The first would make use of the Compute verb and would be written "COMPUTE $R = L * (A - C)/(A + C) ** 2$." The second is a long-hand method:

"Add A and C giving numerator. Subtract C from A giving denominator. Multiply denominator by denominator. Multiply L by Numerator. Divide denominator into numerator giving R."

Nouns

Nouns or names may contain hyphens for readability and ease of reference—for instance, "stores-record" in our earlier example or "movements-item." There are several classes of names in COBOL; among them are:

Data-Names.—A word with at least one alphabetic character, designating any data specified in a data description, e.g. Hours, Name-and-Address-File, Gross-Pay. There are several data levels, since a file may be divided into records which may be divided into smaller groups of data which in turn may be divided into smaller groups, and so on.

Condition-Names.—These specify values which a field can take and for which a test can be made. For example, the field, class of card, can either have the value "issue" or "receipt," so the condition names would be "Issue-card," "Receipt-Card," and the test takes the form "If issue-card . . ." This is instead of saying "If class of card is punched with a 9 then . . ."

Procedure-Names.—Sections and paragraphs in the source program are not numbered for cross-reference but are given names to permit cross-references. For example, one paragraph might be called "Gross-Pay Calculation," and the next "Net-Pay Calculation," and in one paragraph the instruction could be given "Go to Gross-Pay Calculation," and the compiler would know to which paragraph to proceed.

Literals.—A noun identical to those characters represented by the noun—numeric, alpha or alpha-numeric. E.g. "Sequence-Error" to be actually printed in case of error, or "D.C.6B" as a sentinel for comparison.

Qualifiers and Subscripts

Nouns used in COBOL must refer to only one thing—i.e. they must be unique—but adjectival forms can be used to describe names and make them un-ambiguous. This is called "qualification" and there are two types—prefixing and suffixing.

If a prefix is used, the name being qualified is placed last, with the other nouns used adjectivally being placed in descending order of importance, e.g. BRITISH-COMPUTER-SOCIETY CONFERENCE, where CONFERENCE is qualified by B.C.S. as a prefix.

If a suffix is used, the name being qualified is placed first and the other nouns follow in ascending order of importance, with either of the words "OF" or "IN" separating them, e.g.

CONFERENCE OF
BRITISH-COMPUTER-SOCIETY.

Subscripts allow reference to items in a list or table. The name of the required item is followed by its subscript, and the subscript is either written in brackets or is preceded by the word "FOR," e.g.

(*a*) "RATE FOR AGE" (age is subscript)

(*b*) "IF HEIGHT (10) IS GREATER THAN" ((10) is subscript)

COBOL permits entries in tables in up to 3 dimensions, i.e. a rate for an insurance premium may depend on age, weight, and sex. So a valuation statement would be written:

"Multiply Policy-Value by Rate (age, weight, sex)" Brackets are used since "FOR" might be ambiguous.

Conditional Expressions

A condition is a group of words which express something which can be tested to see if it is true or false, and action can then be taken as a result of this test. It is probably easier to give examples than to try and define such matters.

(1) "If pension-contributor, subtract premium from Gross wage," means that the subtraction will be performed if and only if the employee is a pension-contributor.

(2) "If X Y, Move A to B; If Greater Move A to C; If Less Move A to D."
It should be noted that there is no need to repeat the comparison of X with Y as a reference to what quantities "If Greater" and "If Less" apply.

(3) "If Actual-Bonus is not less than Minimum-Bonus, then add Actual-Bonus and Gross-Pay."
This is similar to example (2) except that "then" is used instead of a comma, after the statement of the condition.

(4) "If X equals Y, Move A to B; otherwise Move A to D and also Perform X through Y."
This is a fairly complicated sentence but is essentially of the form: "If C_1 then S_1, otherwise S_2." More complicated sentences can take the form "If C_1 then S_1, and if C_2 then S_2; otherwise S_3," and the meaning or logic of these is not always clear. There is a move in I.C.T. to simplify these expressions by saying that relations should have one subject only and one object only. It is also felt that instead of saying "If A and B greater than C and D, then S_1, otherwise S_2," you would say "If A C, and A D, and B C and B D: then S_1; otherwise S_2." This takes more room but is easier to understand.

Procedure Branches

All source languages must have a means of changing sequences and jumping. COBOL provides this by the two verbs "GO" and "ALTER." An example of the "Go" verb has already been given under "Procedure-Names": at the end of a paragraph named Purchase-Tax one might have the command "GO TO ENTRY-5." If in another part of the source program we wished to change this exit we could write:

"ALTER PURCHASE-TAX TO PROCEED TO ENTRY-3," and the exit from the PURCHASE-TAX paragraph would be altered to "Entry-3" until a further "alter" statement was given. "Entry-3" and "Entry-5" would be names of paragraphs or sections in the same way as "PURCHASE-TAX."

Conclusion

The only way to understand a source language is to use it on actual problems and see if it meets your requirements. Only parts of COBOL have been mentioned here, and the method of DATA DESCRIPTION and the ENVIRONMENT DESCRIPTION have not been touched.

There is a manual in existence for COBOL (Dept. of Defence, 1960), and people interested should try to obtain this, though it is still in a very indigestible form, and by trying it see if this is a useful common language for business applications. I.C.T. will shortly be issuing their more readable version of the common source language.

References

Computing News (1960). *Poobol-Oriented Languages*, Vol. 8, No. 8, p. 11.

Dept. of Defence (1960). *Initial Specifications for a Common Business Oriented Language (COBOL) for Programming Electronic Digital Computers*. April 1960.

Finelli, J. J. (1960). "Development of EDP Units," *The Computer Bulletin*, Vol. 4, No. 1, pp. 12 and 16.

Mitchell, J. W. (1960). "What is Automatic-Coding?" *Automatic Data Processing*, Vol. 2, No. 6, p. 15.